

Tutorial net.objectdays
Erfurt, 9. Oktober 2000

Datenbanken & Java

Gunter Saake
Kai-Uwe Sattler

Institut für Technische und Betriebliche Informationssysteme
Otto-von-Guericke-Universität Magdeburg
{saake|kus}@iti.cs.uni-magdeburg.de

Überblick

- Motivation
- Java und DB-Anwendungsentwicklung
- JDBC
- Embedded SQL
- Java im DB-Server
- Java und OO-Datenbanken
- Objektrelationale Abbildung
- Ausblick

Motivation

- Datenbanken: Basis vieler Anwendungssysteme
  - Internet-Anwendungen, E-Commerce
  - Betriebliche Informationssysteme
- Java als Plattform für Entwicklung solcher Anwendungen
  - Objektorientierung
  - Plattformunabhängigkeit
  - Vielzahl von verfügbaren Bibliotheken



Datenbankzugriff mit Java

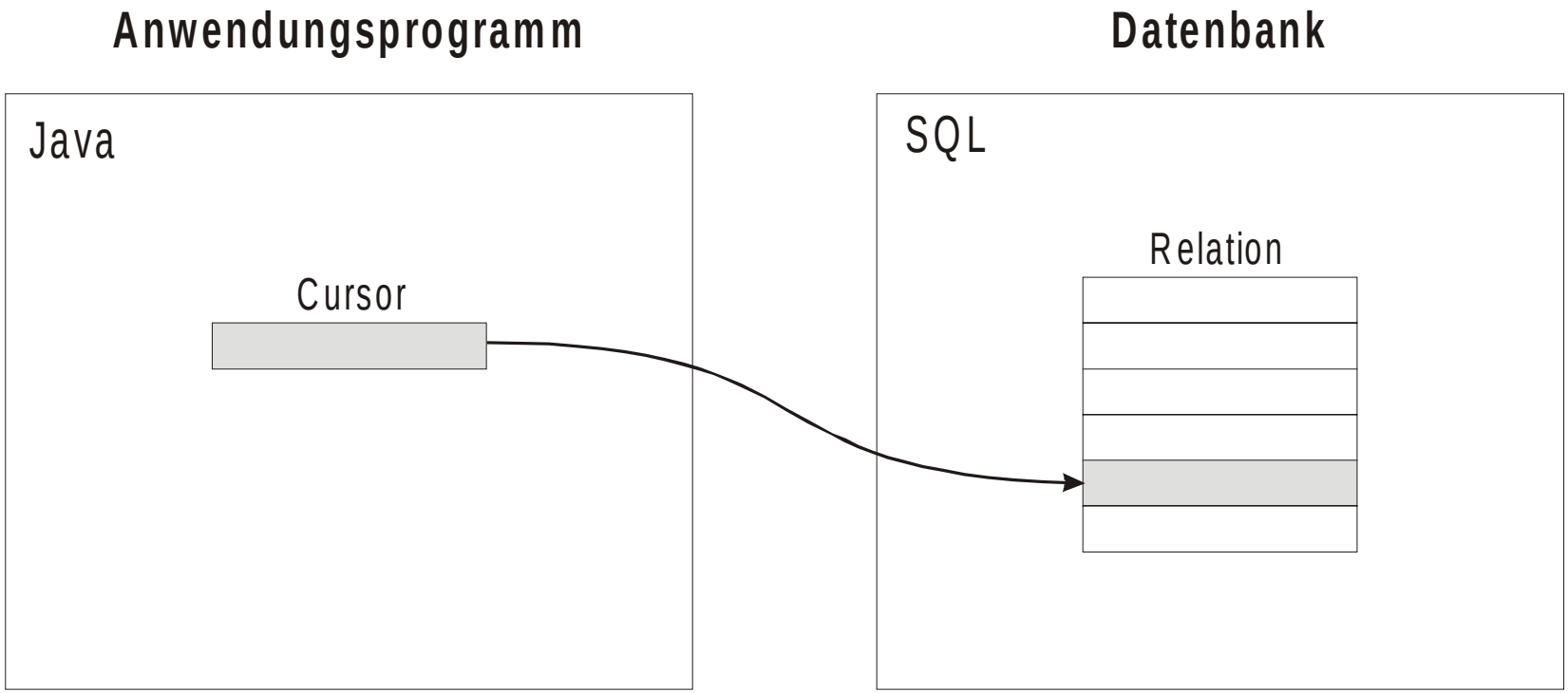
DB-Anwendungsentwicklung

- Kopplung von SQL und prozeduraler Programmiersprache
  - prozedurale Schnittstelle (Call-Level-Interface)
    - ODBC, JDBC
  - Einbettung von SQL
    - Embedded SQL, SQLJ
  - Spracherweiterungen, neue Sprachentwicklungen
    - Persistente Programmiersprachen
    - PL/SQL
- *Impedance Mismatch*: unterschiedliche Datenstrukturkonzepte

Cursor-Konzept

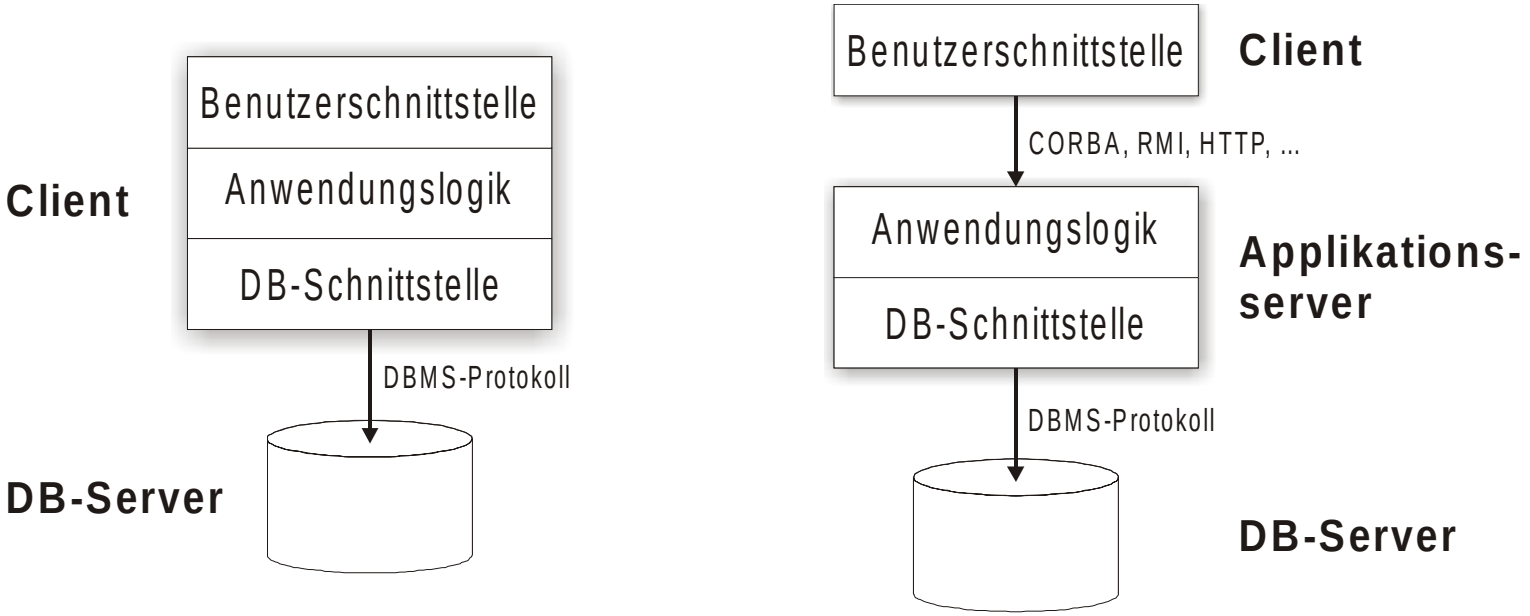
- Motivation
  - SQL: Konzept der Relation als Menge von Tupeln
  - Programmiersprache: Tupel/Objekt als Datenstruktur
- Verbindung Relation - Tupel
  - tupelweiser Zugriff auf Elemente einer Menge (Relation)
  - Navigation über Menge

Cursor-Konzept



DB-Anwendungsarchitekturen

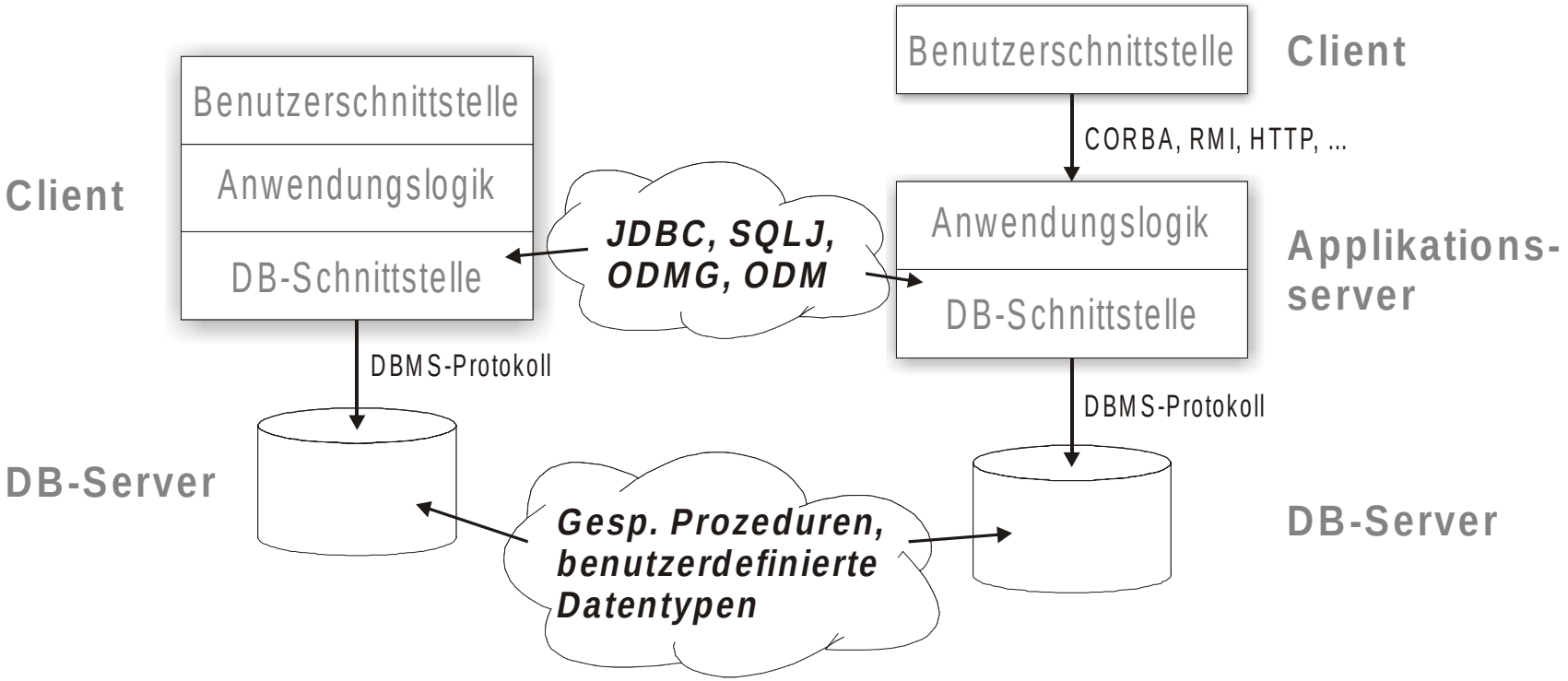
- Schichten interaktiver DB-Anwendungen
  - Präsentation und Benutzerinteraktion
  - Anwendungslogik
  - Datenmanagementfunktionalität



Java und Datenbanken

- Einsatz von Java in DB-Anwendungen
  - auf Client-Seite:
    - Applets, DB-Applikationen
  - auf Server-Seite:
    - DB-Erweiterungen (gespeicherte Prozeduren, benutzerdefinierte Datentypen)
  - mittlere Ebene:
    - Anwendungslogik (JavaBeans, CORBA, RMI)
- Anbindung Java - Datenbank
  - JDBC (Call-Level-Schnittstelle)
  - SQLJ (Embedded SQL)
  - ODMG (ODBMS- und ODM-Zugriff)

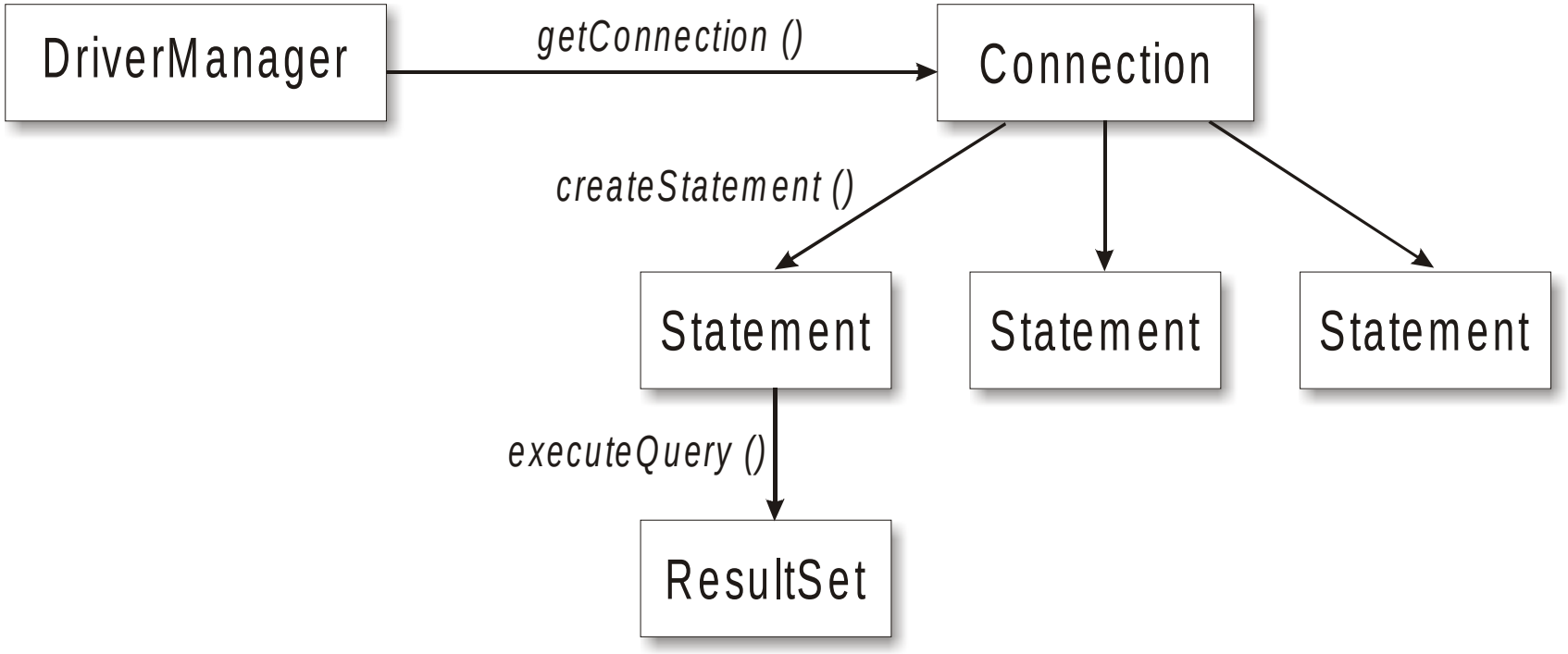
Java und DB-Anwendungen



JDBC: Überblick

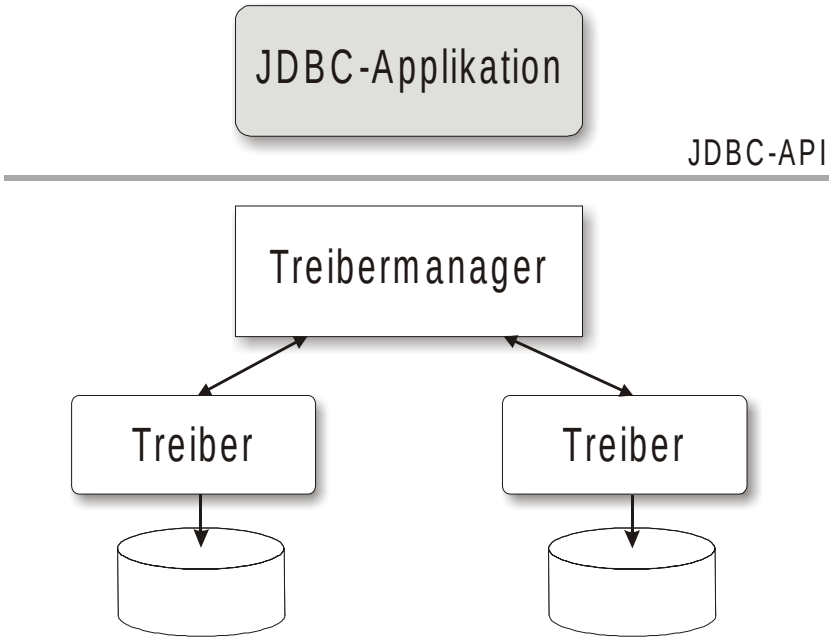
- Datenbankzugriffsschnittstelle für Java
- abstrakte, datenbankneutrale Schnittstelle
- vergleichbar mit ODBC
- Low-Level-API: direkte Nutzung von SQL
- Java-Package **java.sql**
- Klassen
  - **DriverManager**: Einstiegspunkt, Laden von Treibern
  - **Connection**: Datenbankverbindung
  - **Statement**: Ausführung von Anweisungen über eine Verbindung
  - **ResultSet**: verwaltet Ergebnisse einer Anfrage, Zugriff auf einzelne Spalten

JDBC-Struktur



Treiberkonzept

- Treibermanager
- abstrakte Klassen für Datenbankzugriff
- dynamisches Laden von Treibern
- Auswahl des geeigneten Treibers über Verbindungsdaten



Treiberlösungen

- Typ 1: JDBC-ODBC-Bridge
  - Nutzung vorhandener ODBC-Treiber und –Quellen
  - Nur unter Windows
- Typ 2: Native-API-Treiber
  - Nutzung der DBMS-spezifischen API
  - Erfordert Client-Bibliotheken
- Typ 3: JDBC-Net-Treiber
  - Nutzung einer DBMS-unabhängigen Middleware
  - Protokollübersetzung notwendig
- Typ 4: Native-Protokoll-Treiber
  - Direkter Zugriff über DBMS-Protokoll
  - Pure-Java-Treiber

Treiber und URLs

DBMS	Treiberklasse	JDBC-URL
Oracle8	oracle.jdbc.driver.OracleDriver	jdbc:oracle:thin:@*host*:*1521*:*dbname*
IBM DB2	COM.ibm.db2.jdbc.app.DB2Driver	jdbc:db2:*dbname*
MySQL	org.gjt.mm.mysql.Driver	jdbc:mysql://*hos*t/*db*
ODBC	sun.jdbc.odbc.JdbcOdbcDriver	jdbc:odbc:*source*

Ablauf

1. Aufbau einer Verbindung zur Datenbank
   - Angabe der Verbindungsinformationen
   - Auswahl und Laden des Treibers
2. Senden einer SQL-Anweisung
   - Definition der Anweisung
   - Belegung von Parametern
3. Verarbeiten der Anfrageergebnisse
   - Navigation über Ergebnisrelation
   - Zugriff auf Spalten

Verbindungsaufbau

1. Treiber laden
   ```
   Class.forName ("com.company.DBDriver");
   ```
2. Verbindung herstellen
   ```
   Connection con;
   String url = "jdbc:subprotocol:datasource";

   con = DriverManager.getConnection (url,
           "scott", "tiger");
   ```

- JDBC-URL spezifiziert
  - Datenquelle/Datenbank
  - Verbindungsmechanismus (Protokoll, Server-Host und Port)

Anfrageausführung

1. Anweisungsobjekt (Statement) erzeugen
   ```
   Statement stmt = con.createStatement ();
   ```
2. Anweisung ausführen
   ```
   String query = "SELECT titel, preis FROM buch";
   ResultSet rset = stmt.executeQuery (query);
   ```

- Klasse **`java.sql.Statement`**
  - Ausführung von Anfragen (SELECT) mit **`executeQuery`**
  - Ausführung von Änderungsanweisungen (DELETE, INSERT, UPDATE) mit **`executeUpdate`**

Anfrageformen

- Statement
  - Basisklasse für alle Anweisungen
  - Verarbeitung einfacher SQL-Anweisungen ohne Parameter
- PreparedStatement
  - Kapselung einer vorkompilierten Anweisung
  - parametrisierbar
  - Verwendung: wiederholte Ausführung einer Anweisung (mit verschiedenen Parametern)
- CallableStatement
  - Aufruf von gespeicherten Prozeduren mit Parametern

Ergebnisverarbeitung

1. Navigation über Ergebnismenge (Cursor-Prinzip)

```
while (rset.next ()) {
    // Verarbeitung der einzelnen Tupel
    ...
}
```

2. Zugriff auf Spaltenwerte über **`get`*`Type`*-Methoden
   - über Spaltenindex
     ```
     String titel = rset.getString (1);
     ```
   - über Spaltenname
     ```
     String titel = rset.getString ("titel");
     ```

Fehlerbehandlung

- Fehlerbehandlung mittels Exception-Mechanismus
- **`SQLException`** für alle SQL- und DBMS-Fehler

```
try {
    // Aufruf von JDBC-Methoden
    ...
} catch (SQLException exc) {
    System.out.println ("SQLException: " +
        exc.getMessage ());
}
```

Änderungsoperationen

- DDL- und DML-Operationen mittels **`executeUpdate`**
- liefert Anzahl der betroffenen Zeilen (für DML-Operationen)

```
Statement stmt = con.createStatement ();
int rows = stmt.executeUpdate (
    "UPDATE buch SET bestand = bestand-1" +
    " WHERE isbn = '12345' ");
```

Transaktionssteuerung

- Methoden von Connection
  - **`commit ()`**
  - **`rollback ()`**
- Auto-Commit-Modus
  - implizites Commit nach jeder Anweisung
  - Transaktion besteht nur aus einer Anweisung
  - Umschalten mittels **`setAutoCommit (boolean)`**

Transaktionssteuerung

```
try {
    // AutoCommit-Modus ausschalten
    con.setAutoCommit (false);

    // mehrere DML-Anweisungen
    ...

    // COMMIT ausführen
    con.commit ();
} catch (SQLException exc) {
    // Änderungen zurücknehmen
    con.rollback ();
}
```

Vorübersetzte Anweisungen

- **`java.sql.PreparedStatement`**
  - Beschleunigung der Ausführung von Anweisungen durch Vorübersetzung und
  - wiederholten Aufruf mit verschiedenen Parametersätzen
- Parameter in Anweisung durch "**?**" markiert
- Belegung der Parameter mittels **`set`*`Type`***-Methoden

Vorübersetzte Anweisungen

1. Anweisungsobjekt (PreparedStatement) erzeugen

```
String insStmt =
  "INSERT INTO buch VALUES (?, ?)";
PreparedStatement stmt =
  con.prepareStatement (insStmt);
```

2. Parameter setzen

```
stmt.setString (1, "123456");
stmt.setFloat (2, 56.99);
```

3. Anweisung ausführen

```
stmt.executeUpdate ();
```

Aufruf von gespeicherten Prozeduren

- **`java.sql.CallableStatement`**
  - Aufruf von gespeicherten Prozeduren (Stored Procedures)
  - Übergabe von IN-, INOUT- und OUT-Parametern
  - IN-Parameter
    - wie bei PreparedStatement
  - OUT-Parameter
    - Registrierung des SQL-Typs
    - Auslesen der Ergebniswerte mit **`getXXX`**-Methoden

Aufruf von gespeicherten Prozeduren

1. Anweisungsobjekt erzeugen
```
String callStmt = "{ call TestProc (?, ?) }";
CallableStatement stmt = con.prepareCall
(callStmt);
```
2. IN- und INOUT-Parameter setzen
```
stmt.setInt (1, 42);
```
3. Typen der INOUT- und OUT-Parameter registrieren
```
stmt.registerOutParameter (2,
java.sql.Types.FLOAT);
```
4. Anweisung ausführen
```
stmt.executeUpdate ();
```
5. Ergebnisse auslesen
```
double res = stmt.getDouble (2);
```

JDBC-Beispiel

```
try {
        // Verbindungsaufbau
        Connection con = DriverManager.getConnection (
                              url, user, passwd);
        // Anfrageausführung
        Statement stmt = con.createStatement ();
        ResultSet rs = stmt.executeQuery (
              "SELECT titel, preis FROM buch");
        while (rs.next ()) {
              System.out.println (rs.getString (1) +
                    ", " + rs.getDouble (2));
        }
} catch (SQLException exc) {
        // Fehlerbehandlung
        System.out.println (exc);
}
```

Zugriff auf Metadaten

- Metadaten: Informationen über Nutzdaten (Tabellen, Sichten, Spalten, ...)
- **`java.sql.ResultSetMetaData`**
  - Informationen über Struktur des Anfrageergebnisses
- **`java.sql.DatabaseMetaData`**
  - Informationen über Eigenschaften/Fähigkeiten des DBMS
  - Daten aus dem Schemakatalog

Bsp.: Struktur von Anfrageergebnissen

```
// Anfrage ausführen
ResultSet rs = stmt.executeQuery ( ... );

// Metadaten ermitteln
ResultSetMetaData rsmeta = rs.getMetaData ();

// Anzahl der Spalten
int ncols = rsmeta.getColumnCount ();

for (int i = 1; i <= ncols; i++) {
    // Name der Spalte
    String cn = rsmeta.getColumnName (i);
    // Typname der Spalte
    String ctn = rsmeta.getColumnTypeName (i);
    ...
}
```

Neue Features in JDBC 2

- Core API - Bestandteil von Java 2
  - scrollbare und änderbare ResultSets
  - Batch-Updates
  - SQL99-Datentypen
  - Benutzerdefinierte Typabbildung
- Optional API
  - Nutzung von Namensdiensten (JNDI)
  - Connection Pooling
  - Verteilte Transaktionen
  - RowSets

Scrollbare ResultSets

- ResultSet-Typen
  - forward-only: nur vorwärts durchlaufbar, statisch (JDBC 1)
  - Scroll-insensitiv: in beliebiger Richtung durchlaufbar, statische Sicht
  - Scroll-sensitiv: dynamische Sicht: Änderungen anderer Transaktionen werden sichtbar
- Methoden:
  - vorwärts und rückwärts durchlaufen
    - **`next()`**, **`previous()`**
  - direkte Positionierung
    - **`absolute(int)`**, **`relative(int)`**
    - **`first()`**, **`last()`**, **`beforeFirst()`**, **`afterLast()`**

Änderbare ResultSets

- Änderung der Ergebnismenge von Anfragen
- jedoch eingeschränkte Änderbarkeit (ähnlich Änderung von Sichten)
- Methoden
  - Update
    - **`update`*`Type`*`(String, `*`Type`*`)`**
    - **`updateRow()`**
  - Insert
    1. **`moveToInsertRow()`**
    2. **`update`*`Type`*`(String, `*`Type`*`)`**
    3. **`insertRow()`**
  - Delete
    - **`deleteRow()`**

Beispiel: Änderbare ResultSets

```
// änderbares ResultSet erzeugen
Statement stmt = con.createStatement (
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery ( ... );

// Tupel #10  ändern
rs.absolute (10);
rs.updateDouble ("price", 79.0);
rs.updateRow ();

// neues Tupel einfügen
rs.moveToInsertRow ();
rs.updateInt ("order", 42);
rs.updateDouble ("price", 89.0);
rs.insertRow ();
```

Batch-Updates

- Performance-Steigerung durch Zusammenfassen von UPDATE- und INSERT-Operationen
- Methoden
  - Registrieren von Operationen: **`addBatch(String)`**
  - Ausführung: **`executeBatch()`**
- Beispiel:

```
Statement stmt = con.createStatement ();
stmt.addBatch ("INSERT INTO ...");
stmt.addBatch ("INSERT INTO ...");
...
int[] updateCounts = stmt.executeBatch ();
```

SQL99-Typen

- JDBC-Klassen für neue SQL99-Datentypen
  - Binärobjekte: BLOB/CLOB (**`java.sql.Blob`**/**`Clob`**)
  - Kollektionstypen: ARRAY (**`java.sql.Array`**)
  - Strukturierte Typen: CREATE TYPE ... (**`java.sql.Struct`**)
  - Referenztypen: REF (**`java.sql.Ref`**)
- neue **`get`*`Type`***, **`set`*`Type`***-Methoden für **`ResultSet`**, **`PreparedStatement`** etc.

Benutzerdefinierte Typabbildung

- Zuordnung benutzerdefinierter SQL-Datentypen zu selbstgewählten Java-Klassen
- Abbildungstabelle (**`java.util.Map`**) zu JDBC-Connection
- Klasse muß Schnittstelle **`java.sql.SQLData`** implementieren (**`readSQL`**, **`writeSQL`**)
- Nutzung

```
// Registrierung
java.util.Map map = con.getTypeMap ();
map.put ("AddrType", Class.forName ("Address"));
...
while (rs.next ()) {
    // Konvertierung
    Address addr = (Address) rs.getObject (...);
```

RowSets

- Kapselung tabellenähnlicher Strukturen (DB-Relationen, Spreadsheets, etc.)
- serialisierbar → Transfer möglich (Web, PDA)
- Eigenschaften
  - Verbindungsinformationen
  - SQL-Anfrage
- Methoden wie ResultSets
- Implementierung: **`CachedRowSet`**

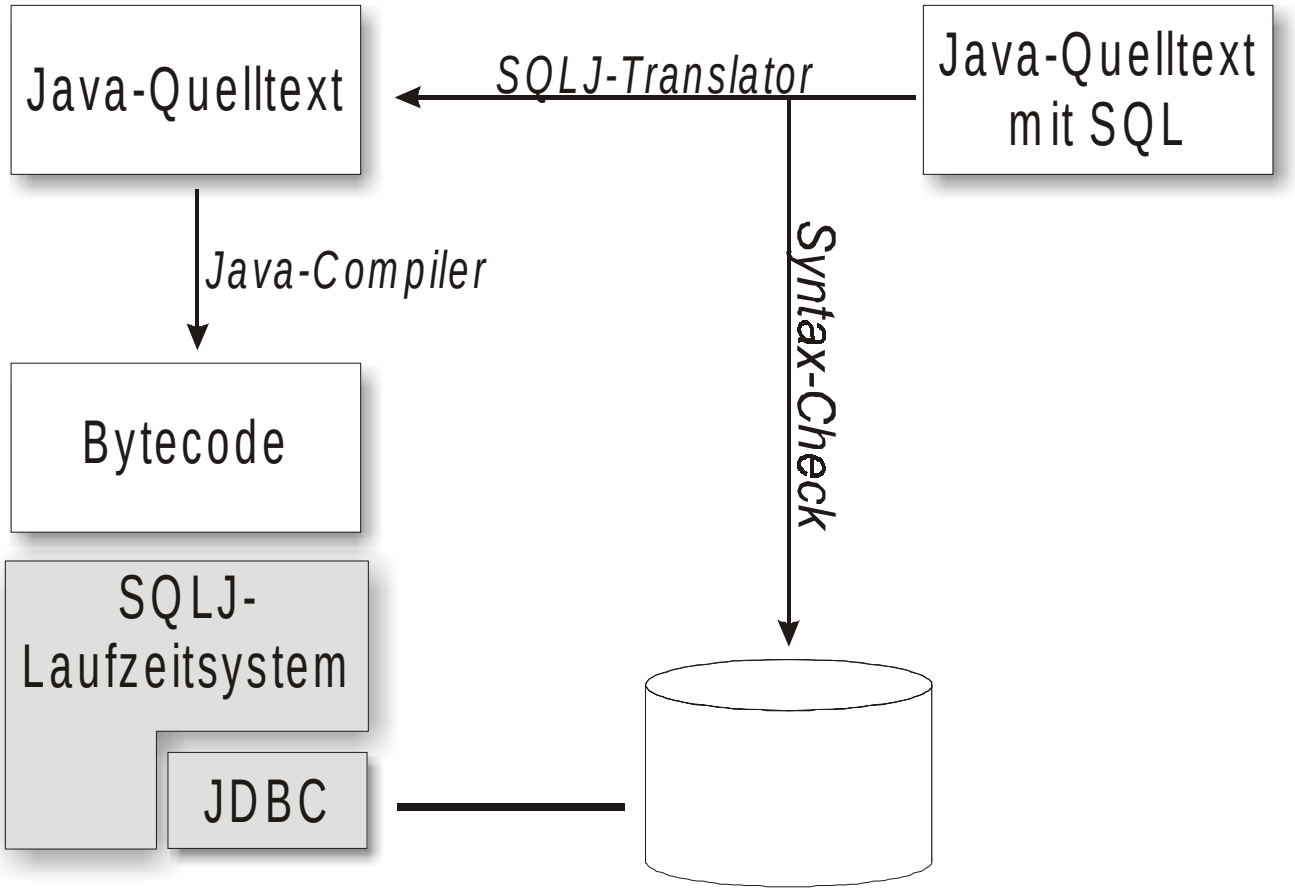
SQLJ

- Industriekonsortium führender DBMS-Hersteller (IBM, Oracle, Sybase, Informix, Cloudscape, Sun)
- Spezifikationen
  - Part 0: Embedded SQL für Java (ANSI-Standard)
  - Part 1: Java Stored Procedures
  - Part 2: Java-Klassen für benutzerdefinierte SQL-Datentypen

Embedded SQL: Überblick

- Einbettung von SQL-Anweisungen in Java-Quelltext
- Vorübersetzung des erweiterten Quelltextes in „echten“ Java-Code durch Translator **sqlj**
- Überprüfung der SQL-Anweisungen
  - korrekte Syntax
  - Übereinstimmung der Anweisungen mit DB-Schema
  - Typkompatibilität der für Datenaustausch genutzten Variablen
- Nutzung von JDBC-Treibern

Embedded SQL: Prinzip



SQLJ-Anweisungen

- Kennzeichnung durch **`#sql`**
- Deklarationen
  - Klassendefinitionen für Iteratoren
- SQL-Anweisungen
  - Anfragen, DML- und DDL-Anweisungen
    ```
    #sql { SQL-Operation };
    ```
  - Beispiel
    ```
    #sql { INSERT INTO buch VALUES
     ("Datenbanken", 59.00, "123456", "MITP")
    };
    ```

Host-Variablen

- Variablen einer Host-Sprache (hier Java), die in SQL-Anweisungen auftreten können
- Verwendung: Austausch von Daten zwischen Host-Sprache und SQL
- Kennzeichnung durch "**`:variable`**"
- Beispiel:

```
String titel, isbn = "123456";
#sql { SELECT titel INTO :titel FROM buch
         WHERE isbn = :isbn };
```

Iteratoren

- Implementierung des Cursor-Konzeptes
- Formen:
  - *benannte Iteratoren*
    - Zugriff auf Spalten des Ergebnisses über Methode mit dem Spaltennamen
  - *Positionsiteratoren*
    - Zugriff über Host-Variablen und FETCH-Anweisung
    - Zuordnung durch Position in der Anweisung

Iteratoren

1. Deklaration des Iterators
```
#sql public iterator BookIter (String titel,
                               double preis);
```
2. Definition des Iteratorobjektes
```
BookIter iter;
```
3. Ausführung der Anweisung
```
#sql iter = { SELECT titel, preis FROM buch };
```
4. Navigation
```
while (iter.next ()) {
   System.out.println (iter.titel () + " " +
                       iter.preis ());
}
```

SQLJ: Beispiel

```
#sql public iterator BookIter (String titel,double preis);
try {
        // Verbindungsaufbau
        Connection con = DriverManager.getConnection (
                                url, user, passwd);
        DefaultContext ctx = new DefaultContext (con);
        DefaultContext.setDefaultContext (ctx);

        // Anfrageausführung
        BookIter iter;
#sql iter = { SELECT titel, preis FROM buch");
        while (iter.next ())
                System.out.println (iter.titel () +
                        ", " + iter.preis ());
} catch (SQLException exc) {
        System.out.println (exc);
}
```

Kontexte

- Verbindungskontext: **`ConnectionContext`**
  - repräsentiert Verbindung (Datenbank, Verbindungsinformation)
  - Definition
    ```
    #sql context MyCtx;
    ```
  - Nutzung
    ```
    ctx = new MyCtx (url, false);
    #sql [ctx] { SQL-Operation };
    ```

Kontexte

- Ausführungskontext: **`ExecutionContext`**
  - Kontrolle der Ausführung der SQL-Operationen, Abfrage des Status der Ausführung
  - Definition und Nutzung

```
ExecutionContext ectx = new
      ExecutionContext ();
ectx.setQueryTimeout (10);
#sql [ctx, ectx] { SQL-Operation };
```

SQLJ-Translator

- Syntaxprüfung
  - Java-Quelltext einschl. Host-Ausdrücke
  - Syntax der SQLJ-Klauseln (nicht SQL !)
- Semantikprüfung
  - Offline-Prüfung der Host-Ausdrücke
    - Einhaltung der Typrestriktionen
  - Online-Prüfung (optional)
    - Analyse der eingebetteten SQL-Operationen (nicht für DDL)
    - Typkompatibilität zwischen Java und DB-Objekten
- Customizing
  - systemspezifische Anpassung (z.B. zur Nutzung spezieller Datentypen oder optimierter Schnittstellen)
  - auch nachträglich möglich über *Profile*

Interoperabilität mit JDBC

- Kombination von statischen und dynamischen SQL
  - Konvertierung zwischen Verbindungskontext und JDBC-Verbindung

```
Connection con =
DefaultContext.getDefaultContext
().getConnection ();
```

  - Konvertierung zwischen Iterator und ResultSet

```
#sql iter = { SELECT * FROM book };
ResultSet rset = iter.getResultSet ();
```

Vergleich

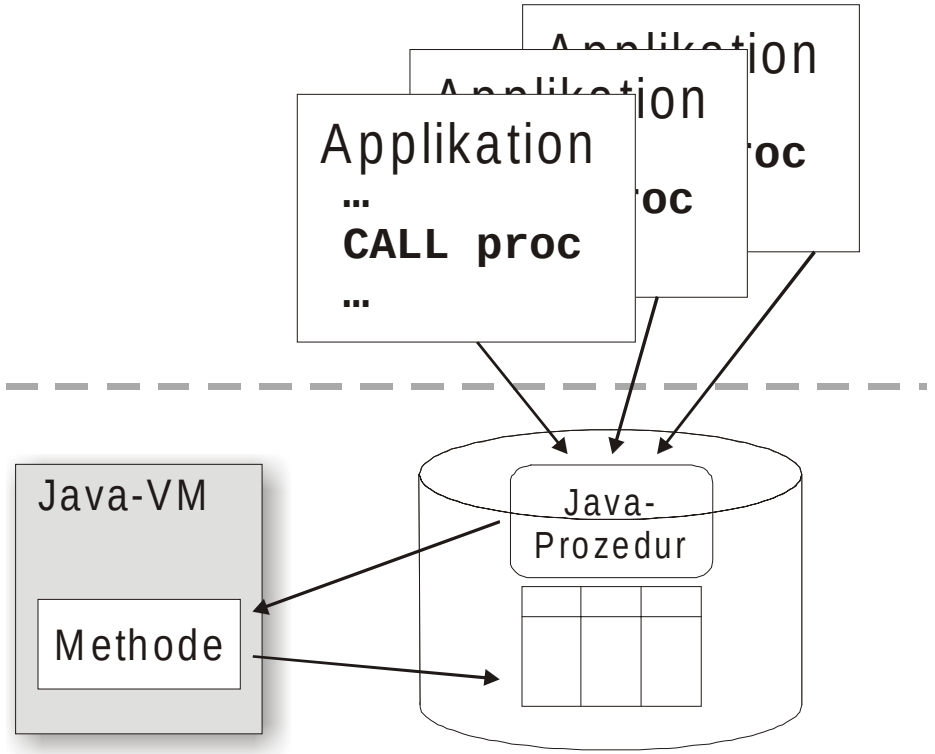
- JDBC:
  - Call-Level-Schnittstelle
  - dynamisches SQL
  - hohe Flexibilität
  - Fehlererkennung zur Laufzeit
- SQLJ Part 0:
  - Embedded-SQL-Lösung
  - statisches SQL
  - Fehlererkennung zur Compile-Zeit

Java im DB-Server

- Java-VM als Teil des DBMS
  - plattformneutrale und sichere Umgebung zur Ausführung von prozeduralen Anweisungen (sicherer Code, Sandbox)
  - Erweiterung des DBMS (Prozeduren, Typen)
- SQLJ
  - gespeicherte Prozeduren/Funktionen
    - Prozeduren, die im DB-Server gespeichert sind, dort ausgeführt werden und damit lokalen Zugriff auf die Daten haben
    - Unterstützung in DB2, Oracle8i, Sybase Adaptive Server, Informix Cloudscape
  - Java-Klassen als SQL-Datentypen
    - benutzerdefinierte Java-Klassen, die als Spaltentypen von Datenbankrelationen verwendet werden können

Gespeicherte Prozeduren

- Vorteile
  - Reduzierung der Netzwerkbelastung
  - Performance-Verbesserung
  - zentrale Verwaltung (Sicherheit, Backup, Recovery, Administration)
  - Unabhängigkeit vom Client-System



Gespeicherte Prozeduren: Ablauf

1. **Implementierung**
   - als statische Methode einer Klasse (DB2: spezielle Klasse **`StoredProc`**)
   - Default-Verbindung zur Datenbank
   - DB-Zugriff über JDBC oder Embedded SQL
   - Beispiel: JDBC

```
public class MyProc {
 public static double price (int orderNo)
   throws SQLException {
  Connection con = DriverManager.getConnection
       ("jdbc:default:connection");
    ...
  }
}
```

Gespeicherte Prozeduren: Ablauf

2. Installation im Server

- als JAR-Archiv
- systemabhängig (spezielles Verzeichnis, DB-Tabelle)
  `sqlj.install_jar('file:~/routines.jar', 'routines_jar', 0)`
- in Oracle8i spezielle Werkzeuge: **`loadjava`**, **`dropjava`**

Gespeicherte Prozeduren: Ablauf

3. Registrierung

- Festlegung des Names, der Parameter und der Zuordnung zur Java-Methode

```
CREATE PROCEDURE myproc (custNo
INTEGER) MODIFIES SQL DATA
EXTERNAL NAME
'routines_jar:Routines.Proc1'
LANGUAGE JAVA PARAMETER STYLE JAVA;
```

- automatisch über Deployment Descriptor

Gespeicherte Prozeduren: Ablauf

4. Aufruf

- über JDBC, Embedded SQL, andere API
- in Anfragen bzw. DML-Operationen
- durch Trigger
- Restriktionen beim Einsatz in DML-Operationen
  - Keine Transaktionskommandos
  - Keine Modifikation von Relationen in **`SELECT`**
  - Kein Zugriff auf zu ändernde Relationen in **`INSERT`**, **`UPDATE`** und **`DELETE`**

Deployment Descriptor

- Automatisierung der Registrierung von Prozeduren sowie der Rechtevergabe
- Textdatei mit **INSTALL**- und **REMOVE**-Abschnitten

```
SQLActions[] = {
 "BEGIN INSTALL
   CREATE PROCEDURE ...;
  END INSTALL",
 "BEGIN REMOVE ..."
}
```

- Markierung im JAR-Archiv durch **MANIFEST**-Datei

Java-SQL-Datentypen

- Verwendung von Java-Klassen als SQL-Datentypen für die Definition von Attributen
- Unterstützung in Sybase Adaptive Server, Informix Cloudscape
- benutzerdefinierter SQL-Datentyp: Klasse mit Attributen und Methoden
- Ablauf
  Implementierung $\rightarrow$ Installation und
  Registrierung $\rightarrow$ Nutzung

Java-SQL-Datentypen: Implementierung

- Klasse mit Schnittstelle **`java.io.Serializable`**
- Attribute zur Repräsentation von SQL-Daten als **`public`** definiert
- zur Ausgabe: Definition der Methode **`toString`**
- Beispiel:

```
public class Address implements
java.io.Serializable {
    public int zip;
    public String city, street;

    public Address () {
          zip = 0; city = street = "";
    }
    ...
}
```

Java-SQL-Datentypen: Registrierung

- Installation als JAR-Archiv über **`sqlj.install_jar`**
- Sybase:
  `INSTALL JAVA NEW FROM FILE 'Address.class';`
- Registrierung über erweitere Form von **`CREATE TYPE:`**
  Zuordnung der Klasse sowie der Attribute und Methoden
- Beispiel:

```
CREATE TYPE Addr EXTERNAL NAME 'addr_jar:Address'
LANGUAGE JAVA (
    zip INTEGER EXTERNAL NAME 'zip',
    METHOD Addr () RETURNS Addr
    EXTERNAL NAME 'Address', ...);
```

Java-SQL-Datentypen: Nutzung

- Verwendung des neuen Datentyps zur Definition von Attributen einer Relation

```
CREATE TABLE customer (
    name VARCHAR(30),
    address Addr);
```

- Initialisierung durch Konstruktor

```
INSERT INTO customer ('Otto', new Addr ());
```

- Zugriff auf Attribute und Methoden mittels SQL99-Syntax

```
SELECT address.zip, address.city FROM customer;
```

- Sybase: spezielle Notation '**>>**'

```
SELECT address>>zip, address>>city FROM customer;
```

Java und OO-Datenbanken

- Java als Definitions- und Implementierungssprache für OODB-Anwendungen
  - direkte Speicherung von Objekten
  - Einsatzbereiche: Datenspeicher für einfache Java-Anwendungen $\rightarrow$ DBMS-Anbindung
- Standard-Binding der ODMG für Java
  - Einführung mit ODMG 2.0
  - Unterstützung durch Mehrzahl der Hersteller

OO-Datenbanken

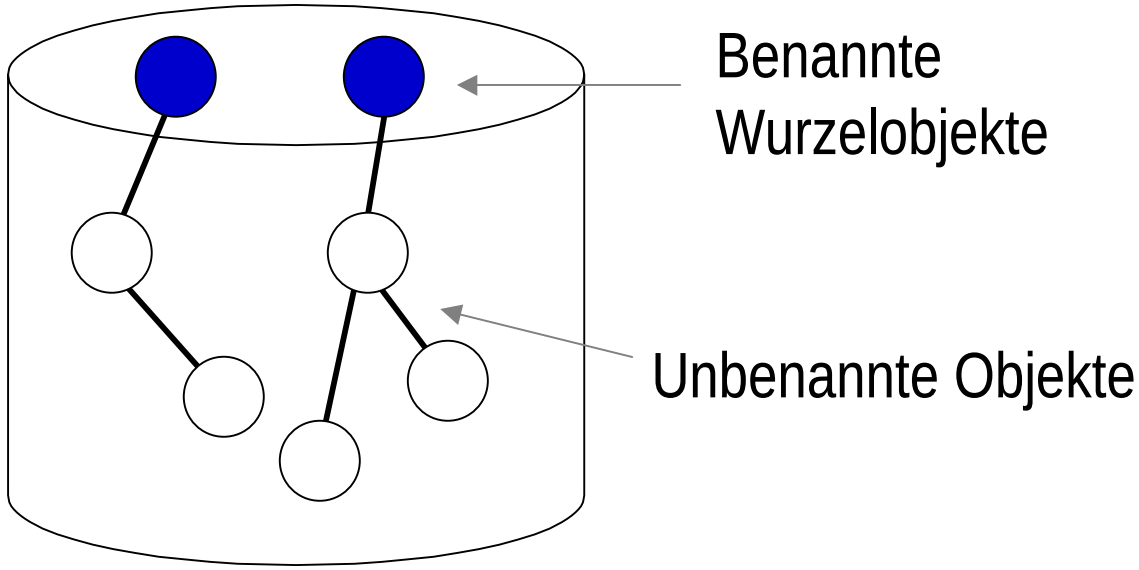
- Unterstützung von Nicht-Standard-Anwendungen
  - komplexe Strukturen (Aggregationen und Assoziationen, strukturierte Eigenschaften)
  - navigierende Zugriffe
  - semantische Modellierungskonzepte (z.B. Spezialisierung)
  - Überwindung des *Impedance Mismatch* (Kluft zwischen Typsystem des DBMS und der Programmiersprache)
  - erweiterte Transaktionsmodelle (lange/kooperative Transaktionen)
- Industriestandard: ODMG 3.0
- Systeme: POET, ObjectStore, Versant, ...

Java-Binding der ODMG

- Prinzip
  - Persistenz durch Erreichbarkeit
  - „persistenzfähige“ Klassen
- Bestandteile
  - Java-ODL: Java als Schemadefinitionssprache
  - Java-OML: DB-Operationen und Manipulation von Objekten als „normale“ Java-Methoden
  - ODMG-API: Klassen für Datenbanken, Transaktionen und Kollektionen
  - OQL-Anbindung: Anfrageunterstützung

Persistenz durch Erreichbarkeit

- Objekt wird persistent, wenn es von einem anderen persistenten Objekt referenziert wird
- erfordert persistente (benannte) *Wurzelobjekte*



Java-ODL

- Java als Schemadefinitionssprache
  - Definition von DB-Klassen in Java
  - keine Spracherweiterungen notwendig
    - Klassen, Vererbung, Attribute, Referenzen und Kollektionen, Methoden
  - Einschränkungen:
    - keine Extensionen und Schlüssel, keine expliziten Beziehungen
    - teilweise mittels zusätzlicher Property-Dateien umgehbar

Java-ODL: Beispiel

```
public class Book {
    // atomare Attribute
    String isbn, title;
    double price;
    // 1:n-Beziehung
    ListOfObject writtenBy;
    // n:1-Beziehung
    Category classifiedIn;

    public Book (String isbn) {
          this.isbn = isbn;
          writtenBy = new ListOfObject ();
    }
    ...
}
```

Generierung persistenzfähiger Klassen

- **Präprozessor**
  - Verarbeitung einer Java-Klassendefinition (Quelltext)
  - Extraktion benötigter Informationen (Vererbungsbeziehungen, Attribute, ...)
  - Hinzufügen DB-spezifischen Codes
  - Kompilieren zu Bytecode
  - Beispiel: POET
- **Postprozessor**
  - Verarbeitung und Modifikation von Bytecode
  - Nutzung existierender Klassen möglich
  - Beispiel: ObjectStore

ODMG-API

- Vordefinierte Klassen für DB-Operationen
  - **`Database`**
    - Öffnen und Schließen einer Datenbank
    - Registrieren und Ermitteln benannter Wurzelobjekte
  - **`Transaction`**
    - Start (**`begin`**), Abbruch (**`abort`**) und Commit (**`commit`**) von Transaktionen
  - Kollektionen **`Set`**, **`Bag`**, **`List`**, **`VArray`**

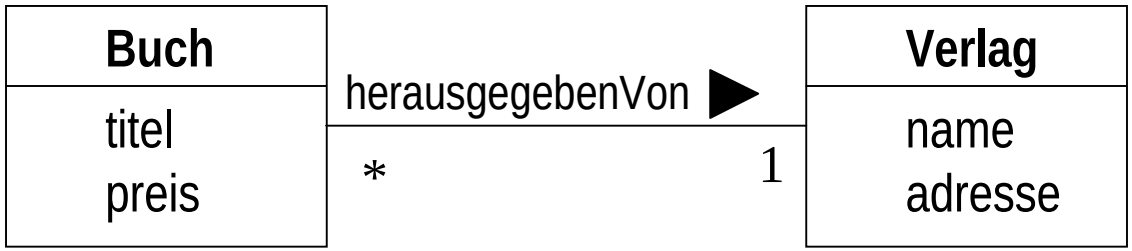
Java-OML: Beispiel

```
// Datenbank öffnen
Database db = Database.open (...);

// Transaktion starten
Transaction tx = new Transaction.begin ();
try {
    // Wurzelobjekt besorgen
    Book b = (Book) db.lookup ("EinBuch");
    // und ändern
    b.setPrice (49.9);
    tx.commit ();
} catch (ODMGException exc) {
    tx.abort ();
}
db.close ();
```

Navigation und Pfadausdrücke

- Navigation in komplexen Werten und Objektstrukturen über *Pfade* → Verfolgen von Beziehungen
- Pfadausdruck: Konstruktion mit Punktoperator ausgehend von Objekt




```
einBuch.herausgegebenVon.name
```

OQL – Object Query Language

- OQL – Anfragesprache der ODMG
- SQL-Subset +
  - Nutzung von Objektidentitäten, Pfadausdrücken, Methoden und komplex strukturierten Werten
  - Anfragen auf Mengen (z.B. Extensionen) und beliebigen Kollektionen
- Anfrage mit relationaler, objekterhaltender und objektgenerierender Semantik
- eingeschränkte Unterstützung in kommerziellen Systemen

OQL-Anbindung

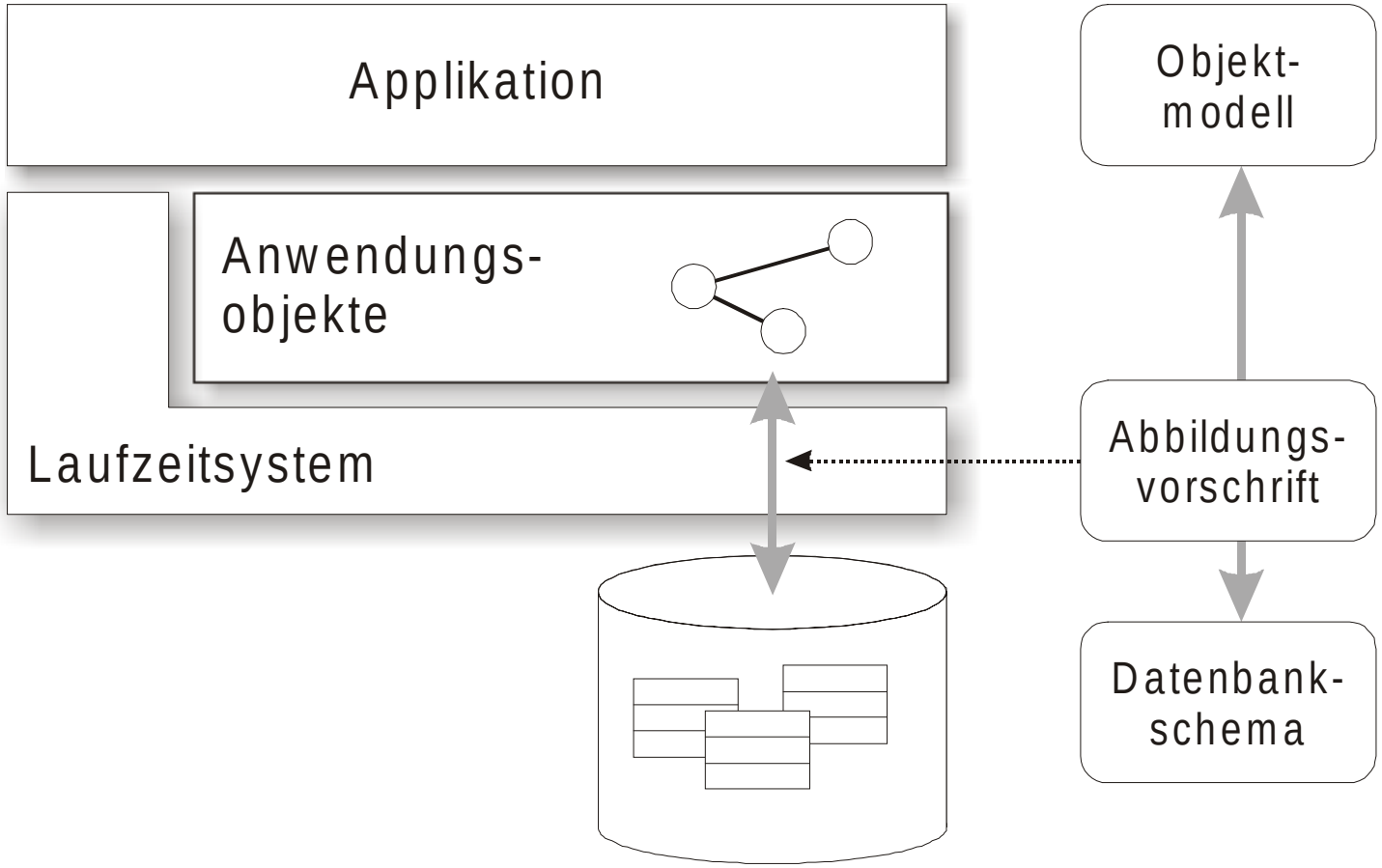
- Klasse **`OQLQuery`**
  - Übergabe der OQL-Anfrage
  - Parametrisierung möglich
  - Ergebnis als Einzelobjekt oder Kollektion
- Beispiel

```
OQLQuery q = new OQLQuery ("SELECT * FROM books");
CollectionOfObject res = (CollectionOfObject) q;
Enumeration e = res.elements ();
while (e.hasMoreElements ()) {
    Book b = (Book) e.nextElement ();
    ...
}
```

Objektrelationale Abbildung

- Idee: Abbildung von Anwendungsobjekten (Java) auf Datenbankrelationen (*Object-to-Database Mapping*)
  - OO auf der Anwendungsebene
  - Nutzung der Vorteile von RDBMS (Sicherheit, Performance, ...)
  - Werkzeugunterstützung
- Richtungen
  - Forward Engineering:
    Java-Klassen $\rightarrow$ Relationenschemata
  - Reverse Engineering:
    Relationenschemata $\rightarrow$ Java-Klassen

Objektrelationale Abbildung: Prinzip



Aspekte

- **Abbildung**
  - Klassen $\leftrightarrow$ Relationenschemata
  - Objektidentifikatoren
  - Attribute und Beziehungen
  - Vererbung (horizontale, vertikale oder typisierte Partitionierung)
- **Laufzeit**
  - statisch vs. dynamisch
  - Verwaltung von Objektpuffern
  - Verfolgen von Referenzen
  - Synchronisation konkurrierender Zugriffe (optimistische vs. pessimistische Verfahren)

Abbildung von Vererbungsbeziehungen

- **Horizontale Partitionierung**
  - Repräsentation der Blätter der Vererbungshierarchie als Tabelle: jede Tabelle enthält alle Attribute
  - vorteilhaft bei abstrakten Superklassen
- **Vertikale Partitionierung**
  - Repräsentation jeder Klasse der Vererbungshierarchie als Tabelle: Tabellen enthalten nur spezielle Attribute der jeweiligen Klasse
  - Anfragen nur auf Superklassen effizient
- **Typisierte Partitionierung**
  - Abbildung aller Klassen in einer Tabelle
  - Unterscheidung durch Typattribut
  - effizient, aber unflexibel (NULL-Werte, Erweiterung)

Synchronisation

- **Pessimistische Verfahren**
  - Verwendung von Sperren → exklusiver Zugriff
  - Einschränkung der Parallelität
- **Optimistische Verfahren**
  - *Annahme*: modifizierende Transaktion kann beendet werden, bevor andere Transaktion das gleiche Objekt ändert
  - Bei Abschluß einer Transaktion: Prüfung, ob Daten inzwischen von anderen Transaktionen verändert wurden (ja → Abbruch, nein → Commit)

Java Blend

- Entwicklungs- und Laufzeitsystem zur objektrelationalen Abbildung (Sun Microsystems)
- Komponenten
  - Mapping-Werkzeug: Definition der Abbildung zwischen DB-Schema und Java-Klassen
    - Java→DB-Modus: Ableitung des Schemas aus Klassen
    - DB→Java-Modus: Generierung von Klassen anhand eines Schemas
  - Präprozessor: Generierung persistenzfähiger Klassen
  - Laufzeitsystem: Funktionen für Laufzeitaspekte und Anfrageausführung

Java-Blend: Mapping-Werkzeug

- **DB → Java-Modus**
  - Auslesen eines existierenden DB-Schemas
  - Analyse der Relationenschemata und Fremdschlüsselbeziehungen (Beziehungen, Vererbung)
  - Generierung einer Default-Abbildung
  - interaktive Anpassung mittels Mapping-Werkzeug
- **Java → DB-Modus**
  - Analyse existierender Java-Klassen, Ableitung eines DB-Schemas
  - Voraussetzungen
    - persistente Klassen implementieren Schnittstelle **`PersistenceCapable`**
    - Beziehungen über Referenztypen oder Kollektionen der Klasse **`DCollection`**

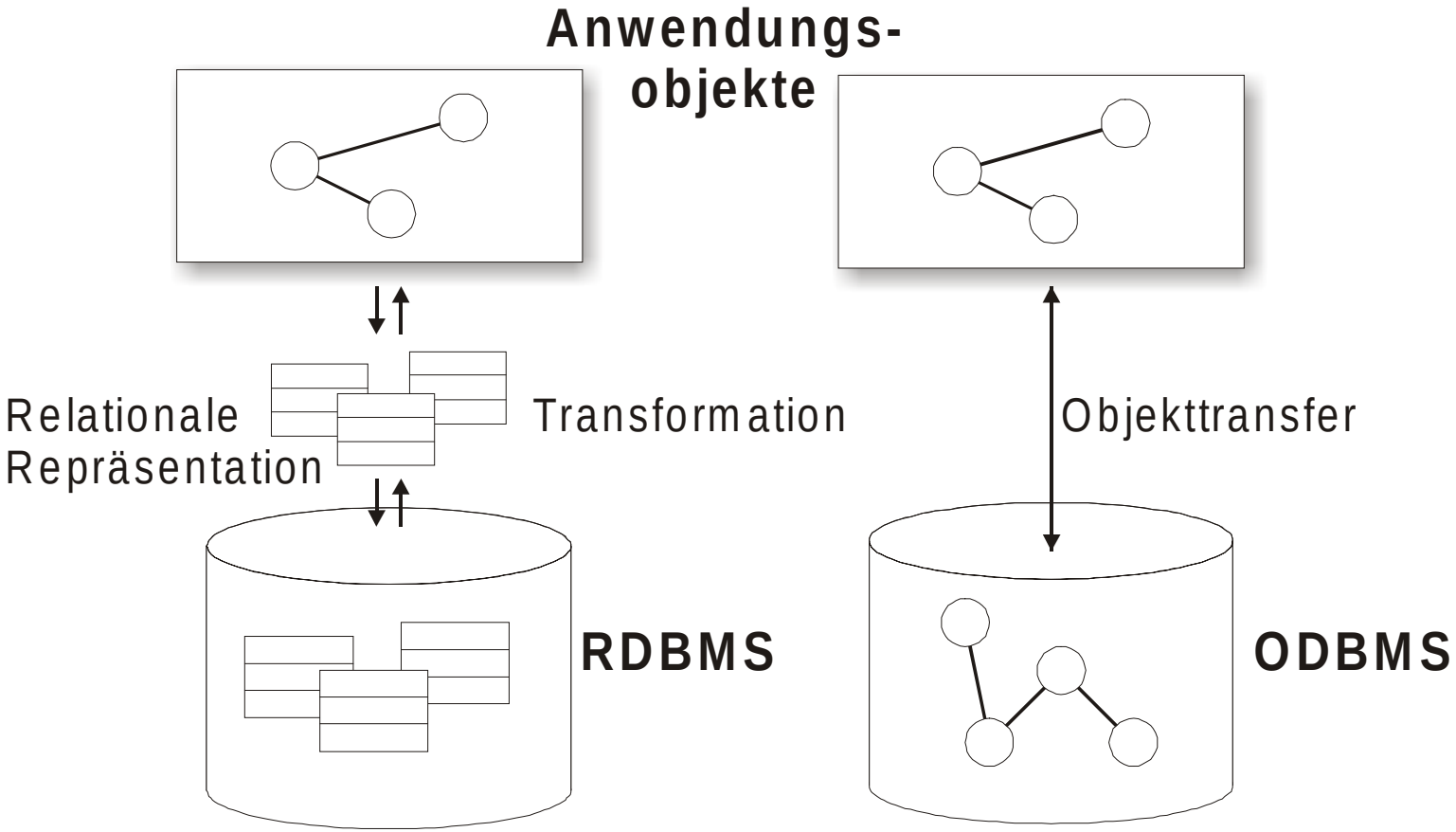
Java-Blend-API

- DB-Zugriff über JDBC-Treiber
- ODMG-konforme Schnittstelle
  - Klassen: **`Database`**, **`Transaction`**, Kollektionen
  - explizite Persistenz: Persistentmachen eines Objektes durch Aufruf von **`makePersistent`**
- Anfragen
  - OQL-Unterstützung inkl. Pfadausdrücke und Methodenaufrufe
  - Transformation in SQL-Anfragen
  - ODMG-konforme Schnittstelle über **`OQLQuery`**

Weitere Systeme

- SQL Object Factory (POET)
- PowerTier (Persistence)
- Relational Object Framework (Watershed)
- ...

Architekturvergleich



Weitere Aspekte

- Komponentenplattformen und Applikationsserver
  - Enterprise JavaBeans
- Transaktionssteuerung
  - Nutzung von verteilten Transaktionen und Transaktionsmanagern über Java Transaction API und XA
  - verteilte Anwendungen, Enterprise JavaBeans
- Web-DB-Anbindungen
  - Servlets, Java Server Pages
- XML-Verarbeitung
  - XML-Parser, Dokument-API (DOM), Werkzeuge

Ausblick

- Java als persistente Programmiersprache ?
  - Orthogonal persistentes Java (PJama/OPJ)
    - Typ-Orthogonalität: Persistenz für alle Objekte
    - Persistenz durch Erreichbarkeit: Persistenz durch transitive Erreichbarkeit von Wurzelobjekten
    - Persistenz-Unabhängigkeit: identischer Programm-code für transiente und persistente Objekte
  - Charakteristika
    - spezielle VM für „Stabilisierung“ des Speichers
    - keine Anfragemechanismen

Ausblick

- Java in der Datenbank
  - Datenbankerweiterungen
  - Java zur plattformneutralen Implementierung von gespeicherten Prozeduren
  - Java-Klassen als SQL-Datentypen
  - Nutzung der Java-Sicherheitskonzepte
- Mobile DB-Lösungen
  - Java als DBMS-Implementierungssprache

Literatur

Saake, Sattler:
Datenbanken & Java: JDBC, SQLJ und ODMG,
dpunkt.verlag, Heidelberg,
Mai 2000, Preis 79,00 DM
ISBN 3-932588-54-1

